Tools for top-down performance analysis of GPU-accelerated applications

Keren Zhou, Mark Krentel, and John Mellor-Crummey

Department of Computer Science

Rice University

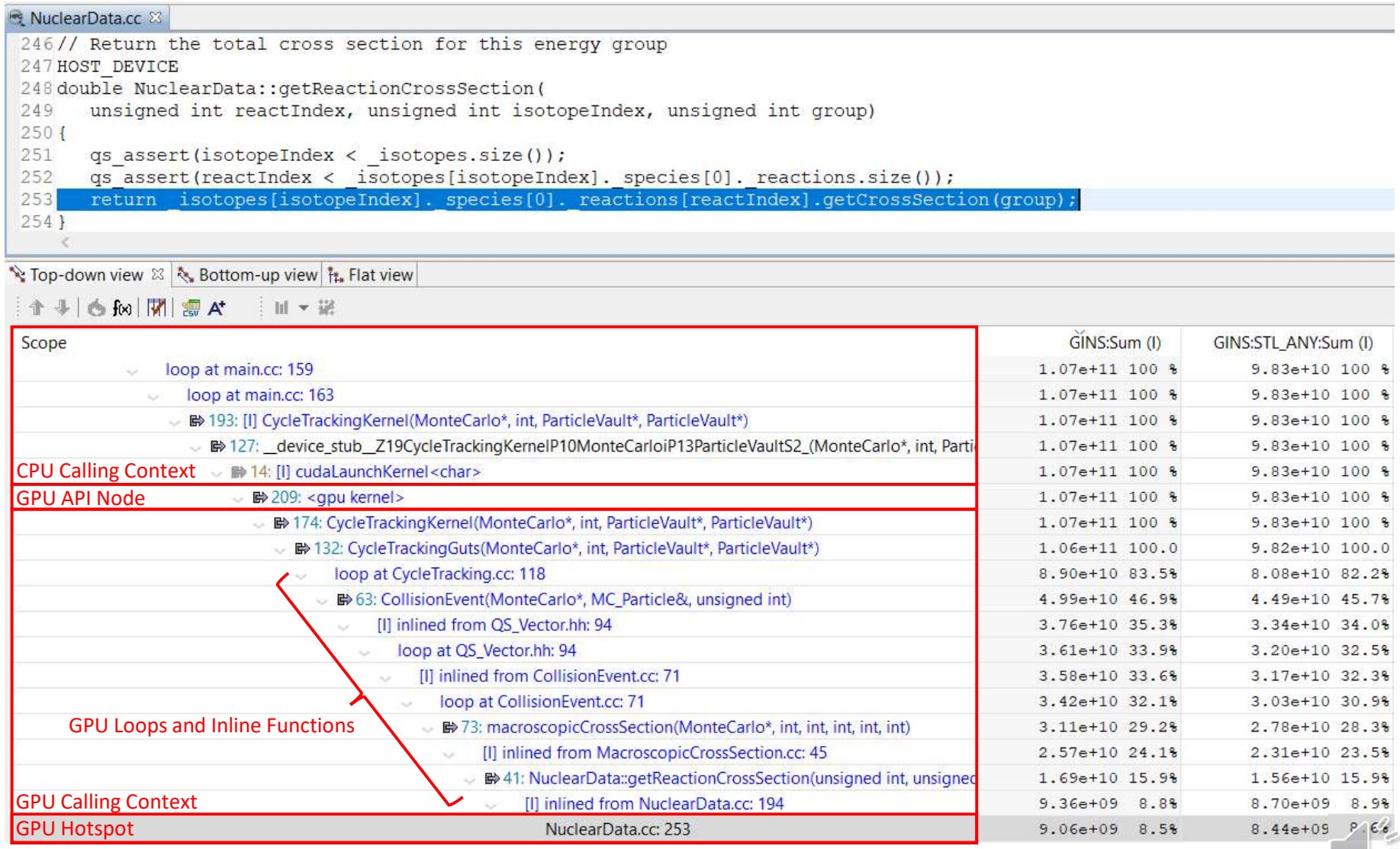
GPU Performance Tools

- Trace view
  - A series of events that happen over time on each process, thread, and GPU stream
- Profile view
  - A correlation of performance metrics with program contexts
- Existing GPU performance tools
  - GPU vendors
    - Nsight Systems, Nsight Compute, nvprof, ROCProfiler, Intel VTune
  - Third parties
    - TAU, VampirTrace, Allinea Map

Problems with Existing Tools

- Existing performance tools are ill-suited for analyzing complex programs because they lack a comprehensive profile view to analyze
  - Sophisticated CPU calling contexts
    - A GPU API (e.g., cudaMemcpy) invoked in different places
  - Sophisticated GPU calling contexts
    - OpenMP Target, Kokkos, and RAJA generate code with many small procedures
- At best, existing tools only attribute runtime cost to a flat profile view of functions executed on GPUs

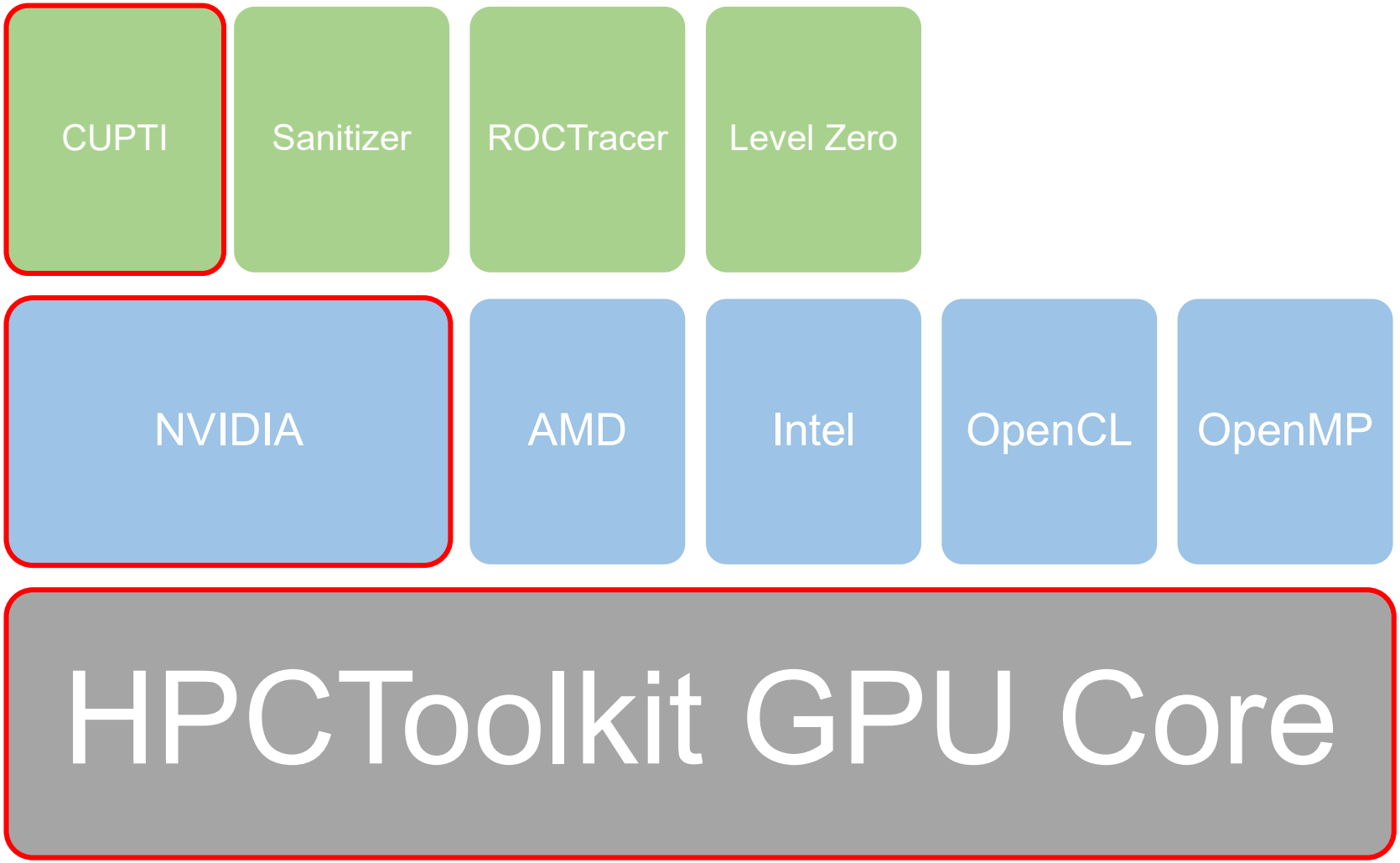
HPCToolkit Profile View



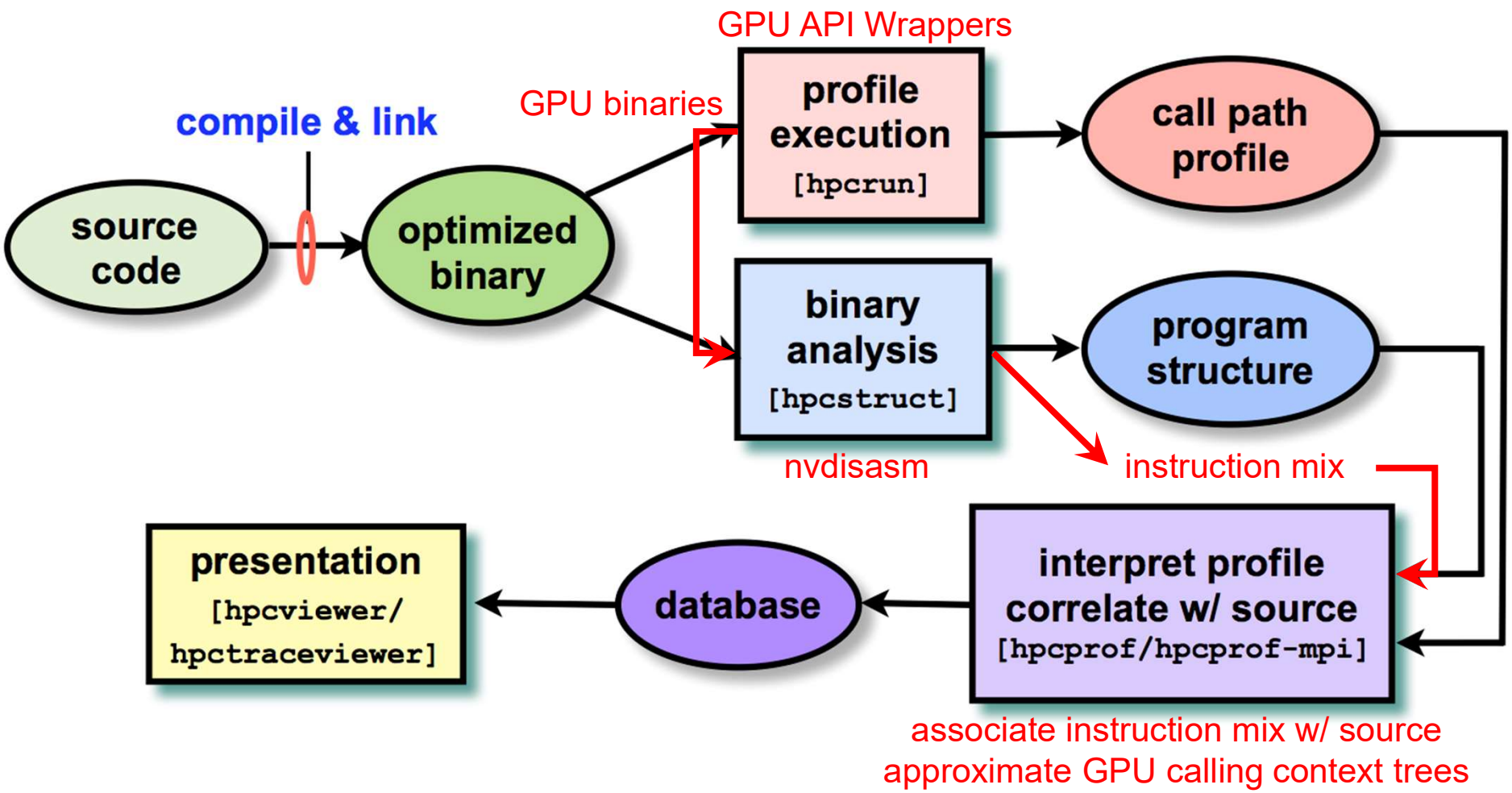
Outline

- Overview
- Performance Measurement
- Performance Metrics Attribution
- Performance Analysis
- Case Studies
- Summary

HPCToolkit Overview



HPCToolkit Workflow



Measurement and Attribution Challenges

- GPU measurement collection
  - Multiple application threads launching kernels to a GPU
  - A monitor thread reads measurements and attributes them to the corresponding application threads
- GPU API correlation in CPU calling context tree
  - Thousands of GPU invocations, including kernel launches, memory copies, and synchronizations in large-scale applications
- GPU metrics attribution
  - Read line map and DWARF in heterogenous binaries
  - Identify loop nests
  - Reconstruct GPU calling context

Outline

- Overview
- Performance Measurement
- Performance Metrics Attribution
- Performance Analysis
- Case Studies
- Summary

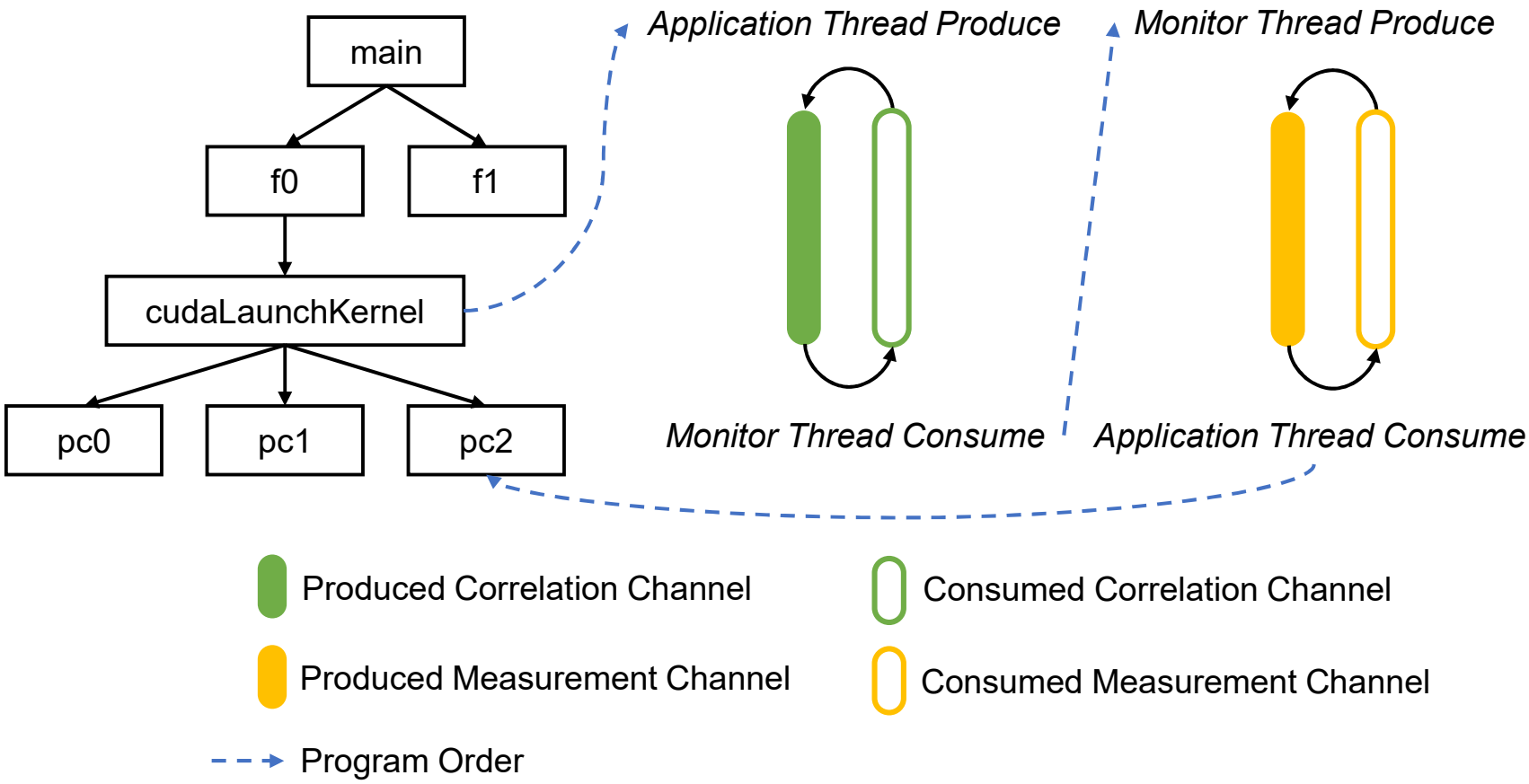
GPU Performance Measurement

- Two categories of threads
  - Application Threads (*N* per process)
    - Launch kernels, move data, and synchronize GPU calls
  - Monitor Thread (*1* per process)
    - Monitor GPU events and collect GPU measurements
- Interaction
  - **Create correlation:** An application thread *T* creates a correlation record when it launches a kernel and tags the kernel with a correlation ID *C*, notifying the monitor thread that *C* belongs to *T*
  - **Attribute measurements**: The monitor thread collects measurements associated with *C* and communicates measurement records back to thread *T*

Coordinating Measurements

- Communication channels: wait-free unordered stack groups
- A private stack and a shared stack used by two threads
  - **Producer PUSH**(*CAS*): push an item on a shared stack
  - **Consumer STEAL**(*XCHG*): steal the contents of the shared stack, push the contents onto a private stack
  - **Consumer POP**: pop an item from the private stack
- Wait-free because **PUSH** fails at most once when a concurrent thread **STEALs** contents of the shared stack

Interactions between the GPU monitor thread and application threads



GPU API Correlation with CPU Calling Context

- Unwind a call stack from each API invocation, including kernel launch, memory copy, and synchronization
- Initial approach:
  - Identify the function enclosing each call site in the call stack using a global shared map
- Problem:
  - Applications have deep call stacks and large codebase
    - Nyx: up to 60 layers and 400k calls
    - Laghos: up to 40 layers and 100k calls

Fast Unwinding

- Refined approach:
  - Memoize common call path prefixes
    - Temporally-adjacent samples in complex applications often share common call path prefixes
    - Employ eager (mark bits) or lazy (tramopoline) marking to identify LCA of call stack unwinds
  - Avoid costly access to mutable concurrent data
    - Cache unwinding recipes in a per thread hash table
  - Avoid duplicate unwinds
    - Filter CUDA Driver APIs within CUDA Runtime APIs

Outline

- Overview
- Performance Measurement
- Performance Metrics Attribution
- Performance Analysis
- Case Studies
- Summary

GPU Metrics Attribution

- Attribute metrics to flat PCs at runtime
  - Relocate each GPU function in a CUBIN so that they do not overlap
- Aggregate metrics to lines
  - Read .debug_lineinfo section if available
- Aggregate metrics to loops
  - nvdisasm –poff –cfg for all valid functions
  - Parse dot files to data structures for Dyninst
  - Use Dyninst ParseAPI to identify loops

GPU Calling Context Tree

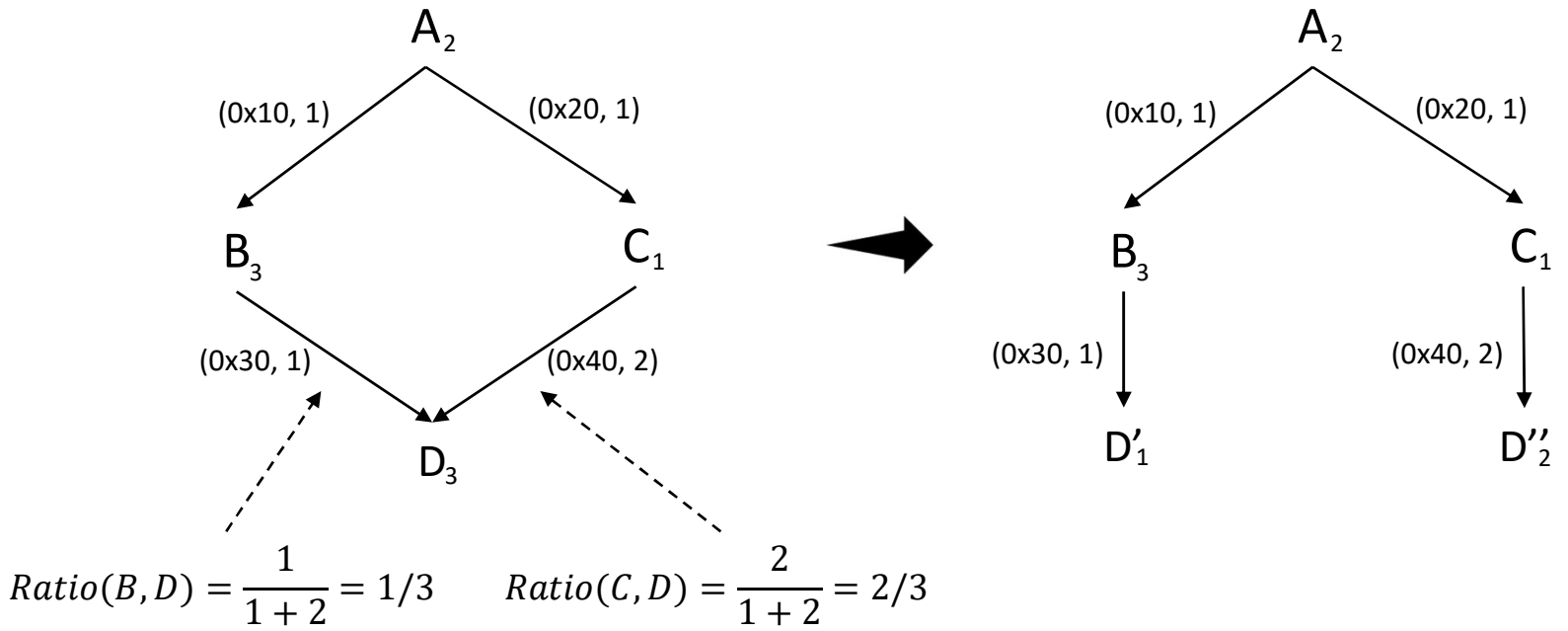
- Problem
  - Unwinding call stacks on GPU is costly for each GPU thread
  - NVIDIA's CUPTI does not provide an unwinding API
- Challenges
  - GPU functions may be invoked from different call sites
  - Need to decide how to attribute costs to each call site
- Solution
  - Reconstruct GPU calling context tree from flat instruction samples and static GPU call graph

Reconstruct Approximate GPU Calling Context Tree

- Construct static call graph
  - Link call instructions with corresponding functions
- Construct dynamic call graph
  - Propagate call instructions to all possible call sites
  - Prune functions with no samples or calls
- Transform call graph to calling context tree
  - Apportion each function's samples based on samples of its incoming call sites
- *See the paper for how we handle recursive calls*

GPU Calling Context Tree Example

- D is split into D’ and D’’ based on (0x30, 1) and (0x40, 2)



$$Ratio(B, D) = \frac{1}{1+2} = 1/3 \qquad Ratio(C, D) = \frac{2}{1+2} = 2/3$$

Outline

- Overview
- Performance Measurement
- Performance Metrics Attribution
- Performance Analysis
- Case Studies
- Summary

CPU Importance

- $\mathrm{CPU}_{\mathrm{IMPORTANCE}}$:

Ratio of a procedure's time to the whole execution time

$$\mathrm{Max}\left(\frac{\mathrm{CPU}_{\mathrm{TIME}} - \mathrm{SUM}(\mathrm{GPU\_API}_{\mathrm{TIME}})}{\mathrm{CPU}_{\mathrm{TIME}}}, 0\right) \times \frac{\mathrm{CPU}_{\mathrm{TIME}}}{\mathrm{EXECUTION}_{\mathrm{TIME}}}$$

Ratio of a procedure's pure CPU time.
If more time is spent on GPU than CPU, the ratio is set to 0

- $\mathrm{GPU\_API}_{\mathrm{TIME}}$:
  - $\mathrm{KERNEL}_{\mathrm{TIME}}$: *cudaLaunchKernel*, *cuLaunchKernel*
  - $\mathrm{MEMCPY}_{\mathrm{TIME}}$: *cudaMemcpy*, *cudaMemcpyAsync*
  - $\mathrm{MEMSET}_{\mathrm{TIME}}$: *cudaMemset*
  - …

GPU API Importance

- $\mathrm{GPU\_API}_{\mathrm{IMPORTANCE}}$

$$\frac{\mathrm{GPU\_API}_{\mathrm{TIME}}}{\mathrm{SUM}(\mathrm{GPU\_API}_{\mathrm{TIME}})}$$

Consider the importance of the memory copy to all the GPU time

- Find which type of GPU API is the most expensive
  - Kernel: optimize specific kernels with PC Sampling profiling
  - Other APIs: apply optimizations based on calling context

Instruction Mix

- Map opcodes and modifiers to instruction classes
- Memory ops
  - class.[memory hierarchy].width
- Compute ops
  - class.[precision].[tensor].width
- Control ops
  - class.control.type
- …

Metrics Derivation

- Problem
  - GPU PC sampling cannot be used in the same pass with metric collection
  - Nsight-compute runs nine passes to collect multiple metrics for a small kernel
- Our approach
  - Derive multiple metrics using PC sampling and other activity records
    - e.g., instruction throughput, scheduler issue rate, SM active ratio
- *See the paper for how we derive metrics*

Outline

- Overview
- Performance Measurement
- Performance Metrics Attribution
- Performance Analysis
- Case Studies
- Summary

Case Studies

- Setup
  - Summit compute node: Power9+Volta V100
  - module load hpctoolkit/2020.03.01
  - module load cuda/10.1.243
- Case studies
  - RAJAPerf Suite
    - Performance benchmark for a template-based GPU programming model
  - Laghos
    - A DOE mini-app that solves the time-dependent Euler equation of compressible gas dynamics
  - Nekbone
    - A subset of kernels from Nek5000, a high-order Navier-Stokes solver based on the spectral element method

RAJAPerf Suite

- Assess the accuracy of our GPU call graph reconstruction
- Compare our call count approximation with exact call counts from NVIDIA's NVBit
  - Compiled with "–G" to avoid function inlining

Test Case	Unique Call Paths	Error
Basic_INIT_VIEW1D_OFFSET	9	0
Basic_REDUCE3_INT	113	0.03
Stream_DOT	60	0.006
Stream_TRIAD	5	0
Apps_PRESSURE	6	0
Apps_FIR	5	0
Apps_DEL_DOT_VEC_2D	3	0
Apps_VOL3D	4	0

Laghos-CUDA

- Pinpoint performance problems in profile view by *importance metrics*
  - CPU takes 80% execution time
    - *mfem::LinearForm::Assemble* only has CPU code, taking 60% execution time
  - Memory copies can be optimized by different methods based on their calling context
    - Use memory copy counts and bytes to determine if using pinned memory with help
    - Eliminate conditional memory copies
    - Fuse memory copies into kernel code

Laghos-CUDA

- Original time: 32.9s
  - 11.3s on GPU computation and memory copies
- Optimized time: 30.9s
  - 9.0s on GPU computation and memory copies
- Overall improvement: 6.4%
- GPU code section improvement: 25.6%

Laghos-RAJA

- Pinpoint synchronization
  - Kernel launch in CUDA is asynchronous, but Laghos uses RAJA synchronous kernel launch
  - Use asynchronous RAJA kernel launch
- Bad compiler generated code with RAJA template wrapper
  - rMassMultAdd<3,4>: RAJA version has 4x STG instructions as the CUDA version. ¼ STG instructions within a loop use the same address.
  - Store temporary values in local variables

Laghos-RAJA

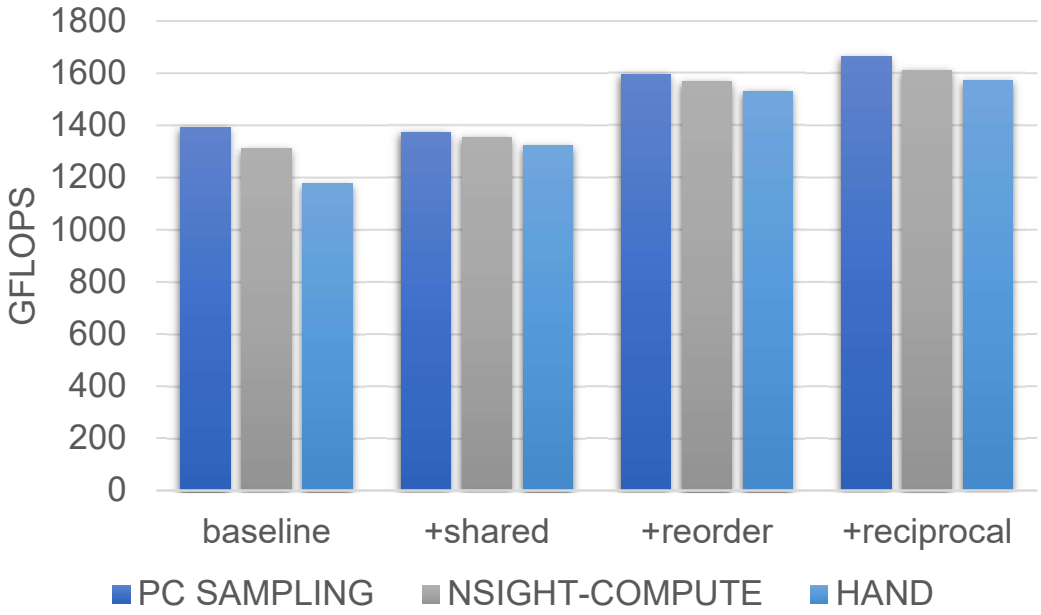
- Original time: 41.0s
  - 19.47s on GPU computation and memory copies
- Optmizied time: 32.2s
  - 10.8s on GPU computation and memory copies
- Overall improvement: 27.3%
- GPU code section improvement: 80.2%

Nekbone

- Associate PC samples with GPU calling context, loops, and lines
- Use instruction mix to provide essential metrics for generating a roofline model
- Problems and optimizations
  - **Memory throttling**: high frequency global memory requests do not always hit cache. +*shared memory*
  - **Memory dependency**: compiler (-O3) does not reorder global memory read properly to hide latency. +*reorder global memory read*
  - **Execution dependency**: complicated assembly code for integer division. +*precompute reciprocal to simplify division*

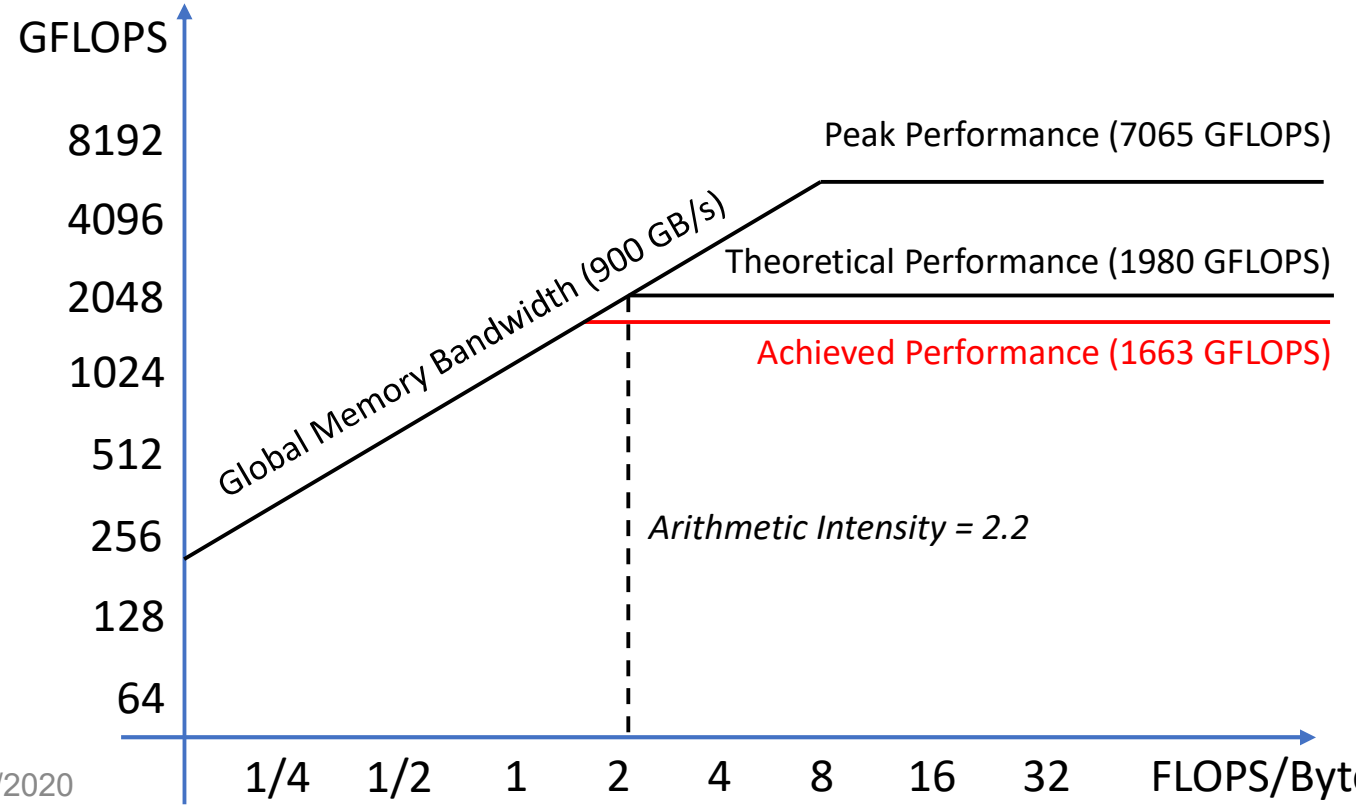
Nekbone Improvement and Error Estimates

- Overall improvement: +34%
- Estimate errors: the first one +8% because of GPU instruction predicates



Nekbone Roofline Analysis

- 83% of peak performance
  - Could obtain +19% by fusing multiply and add on the assembly level



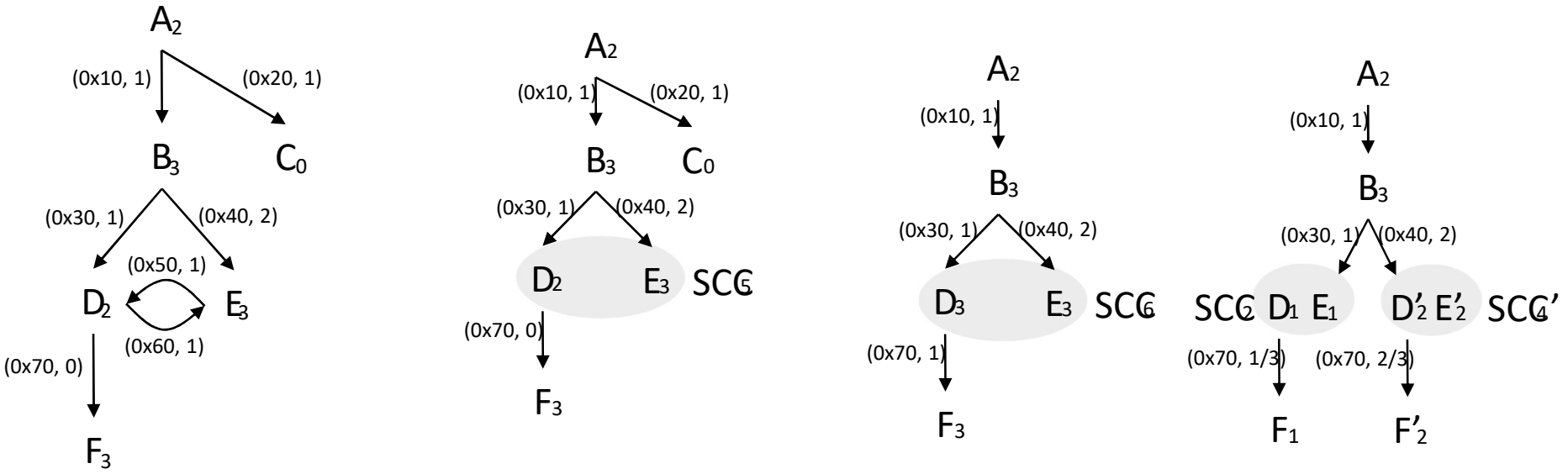
Outline

- Overview
- Performance Measurement
- Performance Metrics Attribution
- Performance Analysis
- Case Studies
- Summary

Summary

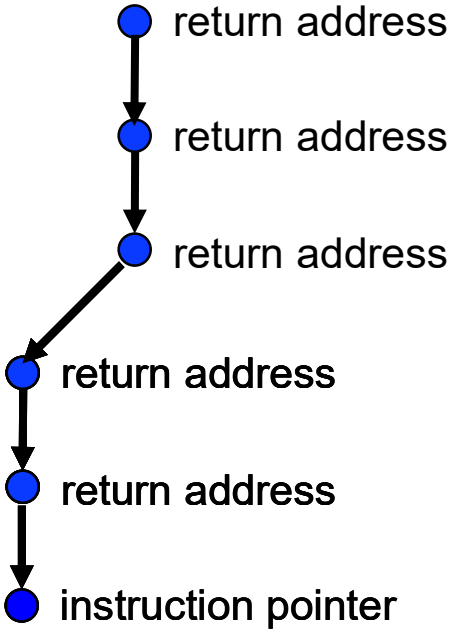
- HPCToolkit pinpoints performance problems for both large-scale applications and individual kernels
- HPCToolkit provides insights for finding problems in compiler-generated GPU code, resource usage, synchronization, parallelism level, instruction pipeline, and memory access patterns
- HPCToolkit collects measurement data efficiently
  - CUDA/10.1
  - Without PC sampling: comparable with nvprof
  - With PC sampling: 6x speedup

GPU Calling Context Tree Example



Memoizing common call path prefixes

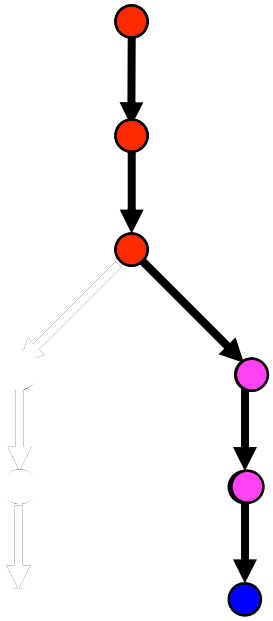
Call path sample



Eager LCA
Arnold & Sweeny, IBM TR, 1999.

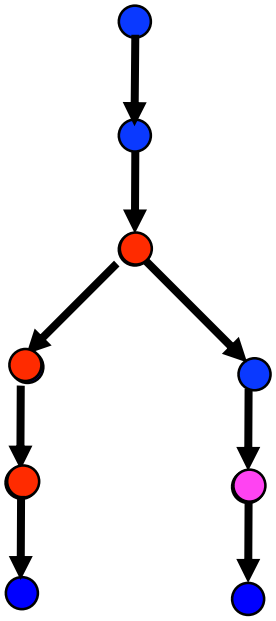
Lazy LCA
Froyd et al, ICS05.

Eager LCA



- **mark frame RAs while unwinding**
- **return from marked frame clears mark**
- **new calls create unmarked frame RAs**
- **mark frame RA during next unwind**
- **prior marked frames are common prefix**

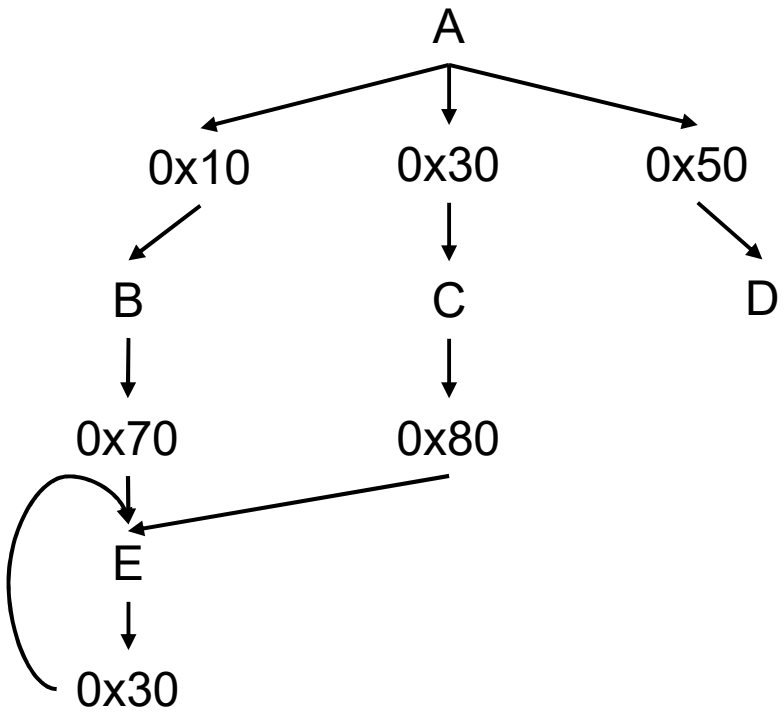
Lazy LCA



- **mark innermost frame RA**
- **return from marked frame moves mark**
- **new calls create unmarked frames**
- **mark frame RA during next unwind**
- **prior marked frame indicates common prefix**

Step 1: Construct Static Call Graph

- Link call instructions with corresponding functions

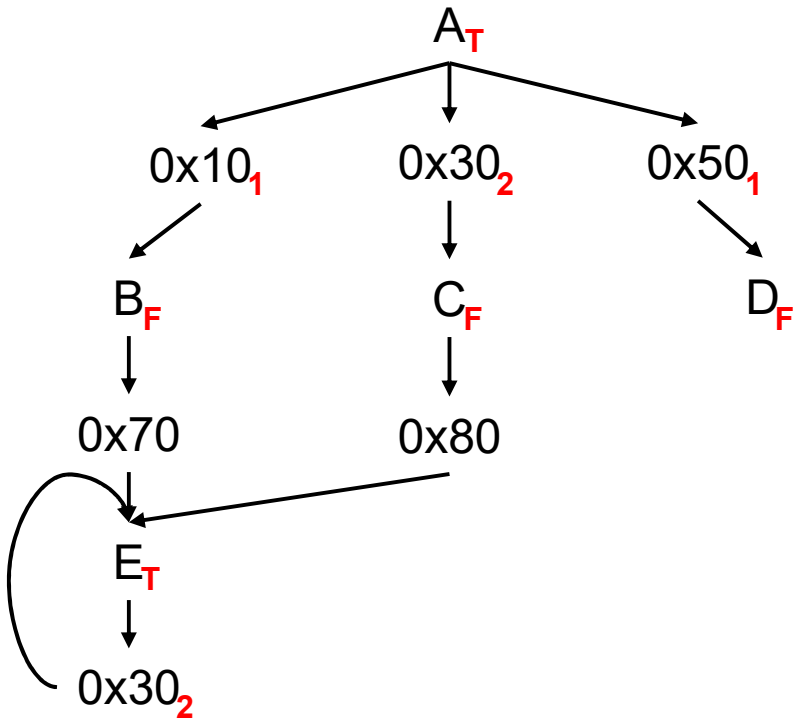


Step 2: Construct Dynamic Call Graph

- Challenge
  - Call instructions are sampled (**Unlike gprof**)
- Assumptions
  - If a function is sampled, it must be called somewhere
  - If there are no call instruction samples for a sampled function, we assign each potential call site one call sample

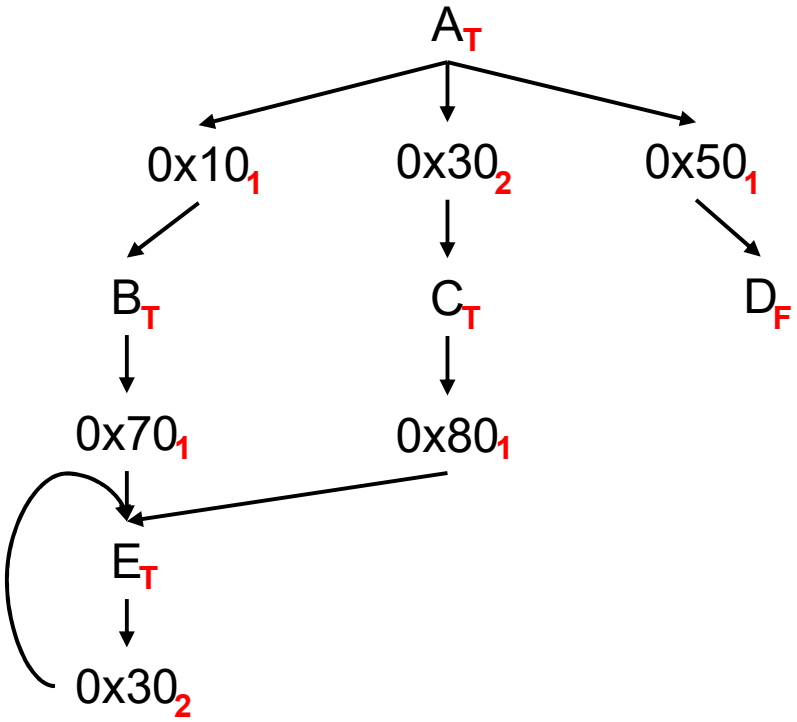
Step 2: Construct Dynamic Call Graph

- Assign call instruction samples to call sites
- Mark a function with *T* if it has instruction samples, otherwise *F*



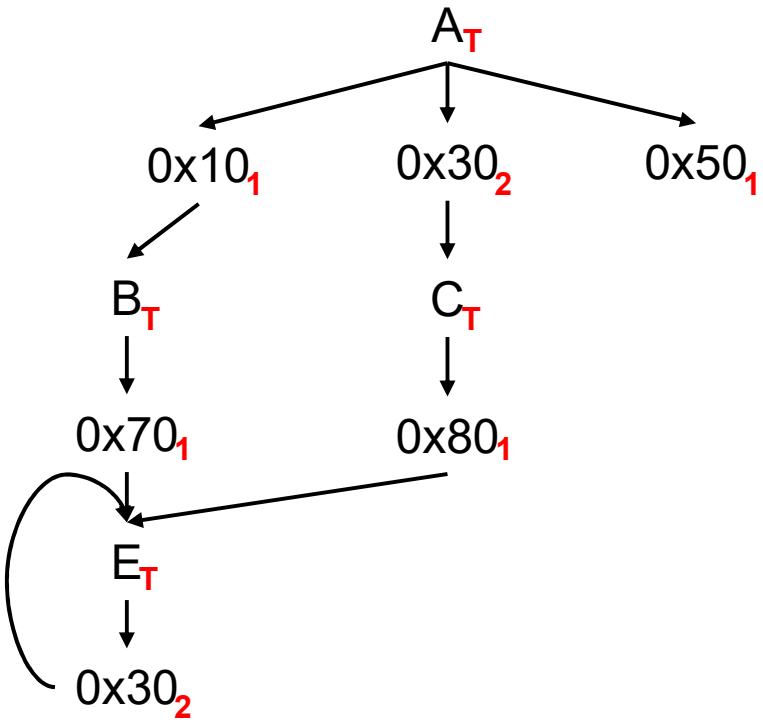
Step 2: Construct Dynamic Call Graph

- Propagate call instructions
  - At the same time change function marks
  - Implemented with a queue



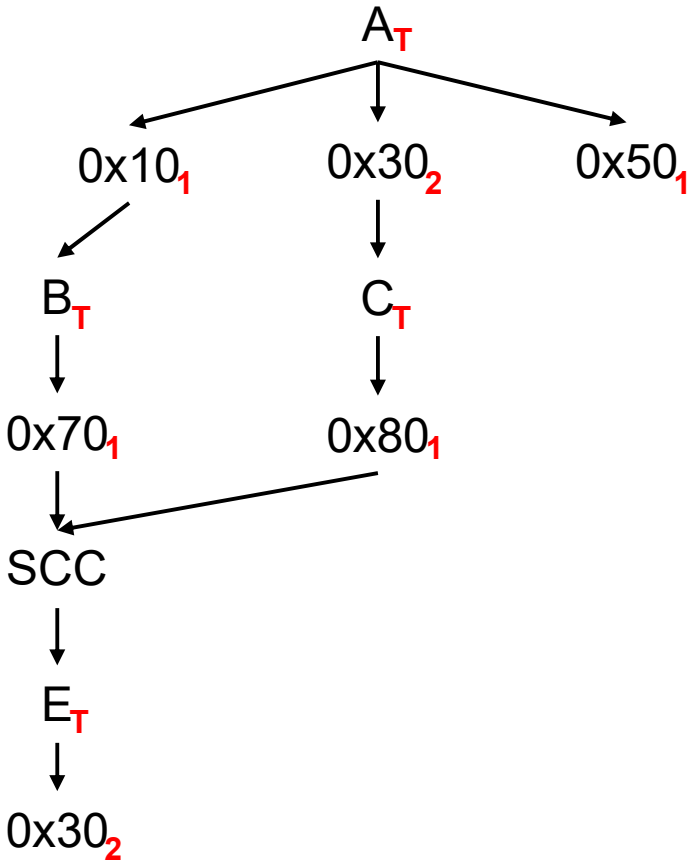
Step 2: Construct Dynamic Call Graph

- Prune functions with no samples or calls
- Keep call instructions



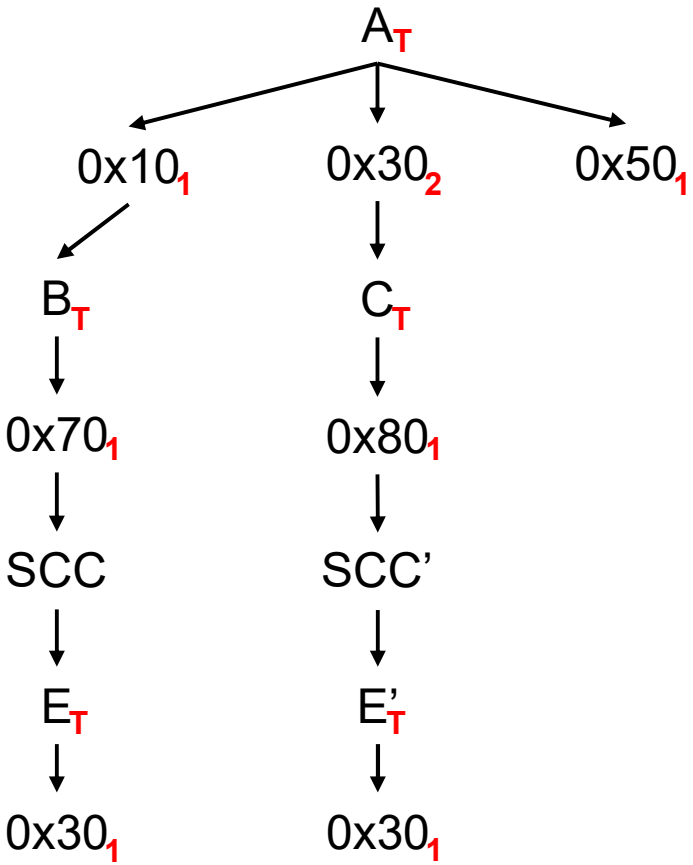
Step 3: Identify Recursive Calls

- Identify SCCs in call graph
- Link external calls to SCCs and unlink calls inside SCCs



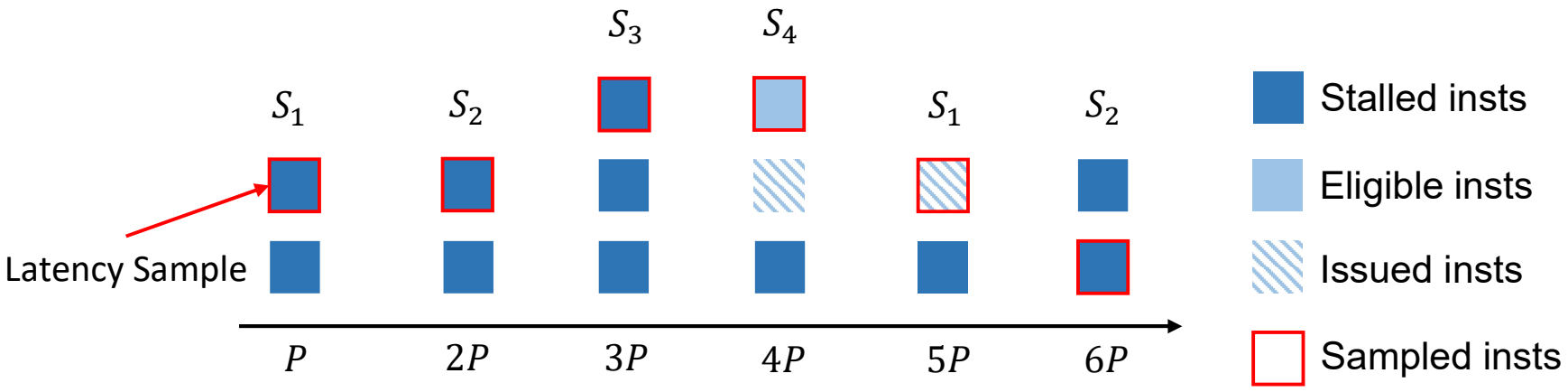
Step 4: Transform Call Graph to Calling Context Tree

- Apportion each function's samples based on samples of its incoming call sites



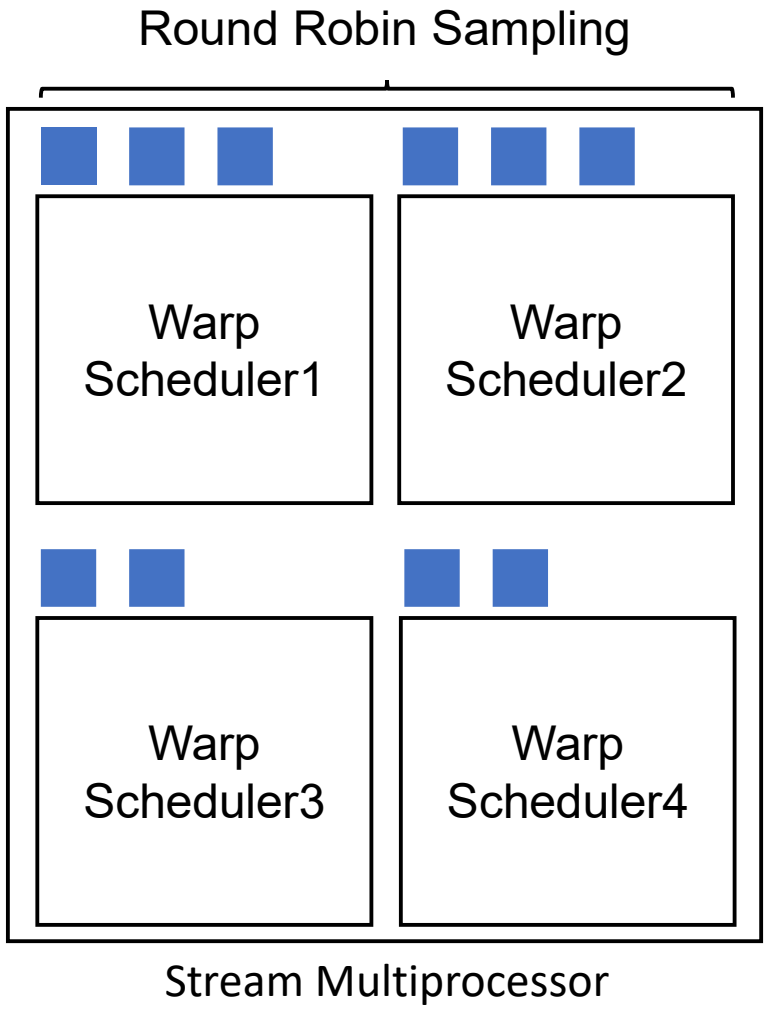
PC Sampling Mental Model

- *Stall reason*: When an instruction is sampled, its *stall reason* (if any) is recorded
- *Latency sample*: If all warps on a scheduler are stalled when a sample is taken, the sample is marked as a *latency sample*



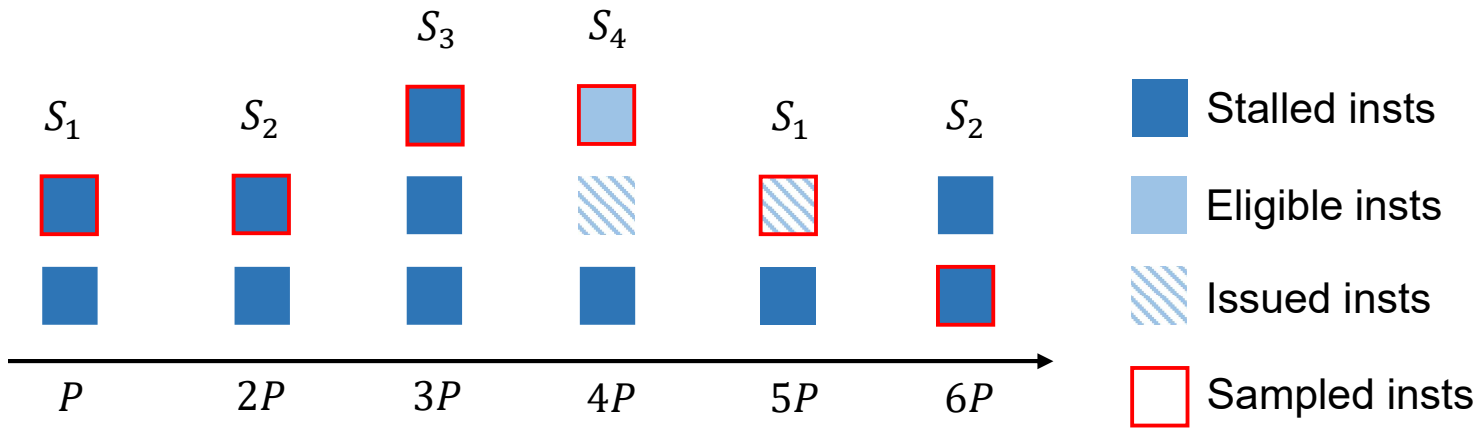
Analysis with PC Sampling

- Each stream multiprocessor is sampled individually
- Active warps are uniformly distributed to warp schedulers
- Samples are taken in a fixed number of cycles
- Volta V100
  - *scheduler_id = warp_index % 4*
  - *16* warp slots on each scheduler



Metrics Derivation Example-IPC

- Total Samples (S): 6
- Latency Samples (S_L): 4
- Issue Rate (I): $\frac{S-S_L}{S} = \frac{2}{3}$
- IPC: $I \times MIN(active\_warps, warp\_schedulers)$



S_1-S_4: schedulers $1P$-$6P$: sampling cycles

Estimate Error

- GPUs are not always running at the maximum clock rate
- Actual average clock rate: $\bar{C}$
- Maximum clock rate: C
- Estimate error: $\varepsilon = \frac{C}{\bar{C}}$